optimization of l'article-in-Cell Codes on RISC Processors

Viktor K. I Decyk

Physics Department, UCkA
Los Angeles, CA 90024
and
Jet Propulsion Laboratory / California Institute of Technology
Pasadena, California 91109
email: decyk@physics.ucla.edu

Steve Roy Karmesin

California Institute of Technology
Pasadena, California 91109
email: ssr@ccsf.caltech.edu

Aeint de Boer

office of Academic (imputing, UCLA
Los Angeles, CA 9(X)24
e-mail: cusgadb@mvs.oac.ucla.edu

Paulett C. Liewer

Jet Propulsion Laboratory/California institute of Technology
Pasadena, California 91109
e-mail: pauly@spaceport.jpl.nasa.gov

Abstract

General strategies are developed to optimize particle-cell-codes written in Fortran for RISC processors which are commonly used on massively parallel computers. These strategies include data reorganization to improve cache utilization and code reorganization to improve efficiency of arithmetic pipelines. Results show performance improvements of 1.4 to 3.4 times can be achieved.

1. Introduction

The recent development of massively parallel processors (MI']%) has offered the promise of great performance, with peak speeds of 30-40 GFlops not unusual. A number of High Performance Computing and Communications (HPCC) projects have been funded in recent years to exploit this new technology for scientific calculations. Some of the authors have been involvéd in one of these, the Numerical Tokamak Project (NTP), funded by the U.S. Dept. of Energy, to model fusion energy devices [1]. One of the important models in this project makes use of Particle-in-Cell (PIC) plasma simulation [2]. PIC codes follow the trajectories of many particles responding to electromagnetic fields that the particles themselves produce. Some of the authors have also been involved in developing PIC codes for other applications, such as an Air Force funded project to model microwave devices.

These codes are challenging to parallelize because they involve two different data structures, particles and fields, that must constantly communicate with one another. Nevertheless, algorithms which effectively parallelize PIC codes have been written [3-4] and they have achieved very good parallel efficiency (typically, 90-%%), meaning that the computation is well distributed on the parallel processors and there is a lot of computation for each unit of communication. Part of this "efficiency" comes from the fact that these codes achieve only a small fraction (generally less than 10%) of the peak speed of the RISC processors commonly used on each node. This is because "efficiency" depends cm keeping the ratio of calculation time to communication time large, and this ratio is enhanced by slow codes!

In order to achieve the goals of the NTP, it was felt that more attention needed to be paid to single node optimization of these parallel PIC codes. This paper reports the strategies we have found useful in improving performance and the results achieved. Of course, since RISC processors are commonly used on workstations, any improvements we can make would also be useful there. Furthermore, since many of the optimization strategies are quite general, they may also be useful in other types of calculations. We measure here only the single node performance of algorithms. The issues involved with parallelization are described elsewhere [3-4].

In this paper we investigate ten different strategies for optimization of Fortran code on the RISC processors of the Intel Paragon, Gay T3D, and IBM S1'2. By combining the various strategies, we were able to achieve overall improvements in PIC code performance of 1.4 to 3.4 times, depending on the processor. The strategies involved changes to the Fortran code only; no assembly language code was used.

11. RISC Processors

RISC (Reduced instruction Set Computer) processors have quite different strengths and weaknesses compared to the high performance vector architectures many in the scientific community are familiar with. We describe here some of the key attributes of RISC processors and how they impact code design ant{ optimization strategies.

One way the RISC processors achieve high performance is through the use of pipelined arithmetic. This means that a multiply operation, for example, is broken down into a number of parts, and the processor can be working on different parts of different multiplies at the same time. Although it may take some number of clock periods to get the first multiply (typically < 8), once the pipeline is filled, one result can be produced per clock subsequently. Of course, other operations can begin in successive clock periods only as long as they do not use as an input the result of an operation in progress. The floating point units take their inputs and store their results to registers, and there are usual] y between 16 and 64 of them, each able to hold a single floating point number. When operating only on data in registers and when there are no data dependencies among the operations, the RISC processors considered here operate in the 75 to 266 MFl ,01's / Second range.

Vector processors work on the same principle, but they have special instructions for performing many multiplies or adds at once, which then fill the pipelines. To take advantage of vector architecture, the programmer must "vectorize" his code, that is, write his or her code in such a way that the compiler can recognize when such multiple (i.e., vector) instructions can be used. In RISC architecture, the pipelines are filled by the compiler with whatever instructions it can find. This is made somewhat easier by the fact that the pipelines on RISC processors are typically much shorter than on vector processors.

Useful computations cannot be done with only data in registers of course, so data must be loaded and stored from memory, and it is here that another complex feature of using RISC processors is found. To keep the cost of memory down, RISC processors use large amounts of slow-but-cheap DRAM (Dynamic Random Access Memory) and smaller amounts of fast-but-expensive SRAM (Static Random Access Memory). The SRAM serves as a cache, that is, a staging memory between the fast CPU and the relatively slow main memory. To make the hardware fast the algorithms used to decide what parts of memory to keep in cache are very simple. The cache is composed of a number of cache "lines" of adjacent words (typically 4-32). Although fetching one word from main memory to cache can be relative slow (typically tens of clock periods), the adjacent words in memory are fetched to the cache at the same time, and can then be loaded to registers directly from cache. Aside from the number of clock cycles it takes for a given load or store to complete, this staging is handled transparently by the RISC processor's memory management unit. This transparency allows the same executable to run correctly on machines with different cache sizes and layouts.

Since the cache is generally small, different locations in main memory can be mapped to the same location in cache memory. If two operands needed in rapid succession in a calculation are mapped to the same cache location, performance wil be degraded. Not only will the processor be forced to wait while those cache lines are replaced, but reuse of the other elements of the cache line will also be inhibited. Although details depend on the processor, it is general] y the case that if memory references are close together, it is less likely that they will occupy the same cache location. In contrast, the memory in some vector processors such as the C90, is organized so that the stride through memory is more important than location in

memory.

There are two main ways to achieve good performance on RISC processors. One way is based on effective use of cache. The other way is through the effective use of pipelined arithmetic. In the remainder of this paper we will investigate how this can be done for PIC codes by modifying the Fortran code.

111. Structure of Particle-in-Cell Codes

We will use a well bench-marked electrostatic PIC code [4 1 as our protot ype algorithm because it is conceptually simple while demonstrating the techniques involved. The time step for an electrostatic PIC code has three main parts: deposit, field solve, and push. In the deposit step, charge is deposited from the particles to a grid to find the charge distribution. Each particle deposits charge to several nearby grid points and many particles deposit to each grid point. In the field solve step, a Poisson equation is solved on the grid to find the electric field. In the push step, the electric field is interpolated to the particles to find their acceleration. Each particle reads the electric field from several nearby grid points and each grid point is read by many particles. These three main parts are repeated for as many time steps as desired. The deposit and push steps are characterized by having memory references that are effectively random and change from one time step to the next, making these steps challenging for computer memory systems that are optimized for regular reference pat terns. Let us go over each of these steps in more detail.

in one dimension with linear interpolation the deposit consists simply of finding the two nearest gridpoints to the particle's location and depositing charge which is proportional to the separation from the grid point, as follows:

```
dimension q(nx)
n =  int(x)
dx =  x  -  float(n)
q(n)  =  q(n)  + qm*(1  -  dx)
q(n+1) = q(n+1) i qm*dx
```

where x, qm are the position and charge of the particle, nx is the number of grids, and q is the charge density function. Note that an important part of this calculation is an indirect memory reference based on n which is accumulating into the array q (scatter operation). Each successive particle could reference the same or different memory locations. The charge is accumulated into each location, so the order in which the particles are treated is not important (we ignore roundoff errors).

A Poisson's equation is then solved to give the electric field E on the grid points:

$$\nabla \bullet E = 4\pi q$$

This is done using FFTs. Since typically less than 5% of the CPU time is spent solving for the field and FFTs have been extensively studied, it will not be discussed further here.

The last step consists of accelerating the particles in the calculated electric field to their new positions, using Newton's 1a w, F = ma. The algorithm used here is a time-centered leap-frog scheme, which in one dimension consists of:

```
vx(t+ dt/2) = vx(t-dt/2) + qtm*f(t)
x(t+dt) = X(t) + vx(t+dt/2)*dt
```

where vx is the particle velocity, dt is the time step, qtm = (q/m)*dt is the charge-to-mass ratio times the time step, and f(t) = fx(x(t)) is the electric field at the particle's current position, which is found by interpolation from the field at the two nearest grid points:

```
dimension  fx(nx)
n = int (x)
dx = x - float(n)
f = fx(n)*(1 - dx) + fx(n+1)*dx
```

Note that an important part of this calculation is an indirect memory reference in the array fx which is random (gather operation). Once again, successive particles can reference the same or different locations in the array fx, and the order in which particles are treated is unimportant.

in this paper we will be using quadratic interpolation, because it more closely resembles the algorithm used in the Numerical Tokamak Project. in one dimension that means the interpolation involves the three grid points nearest the particle position, as illustrated in the following charge deposit example:

```
n = int(x+ .5)
dx = x - float(n)
q(n-1) = q(n-1) + .5*qm* (.5 - dx) **2
q(n) = q(n) + qm* (.75 - dx**2)
q(n+1) = q(n+1) + .5 *qm *(.5 + dx)**2
```

with a similar scheme for the acceleration.

IV. Test Case Description

The standard benchmark case uses a two dimensional grid of 64 x 128 cells, with 327, 680 particles, which gives about 40 particles per cell on average. The particles are initially sorted by position, but randomize as the run proceeds. All results are calculated after the particles have become randomized, and were done with 64 bit precision.

Three parallel computers will be used in the evaluation, the Intel Paragon, the Cray T3D, and the IBM S1'2. These are the main machines of interest to the Numerical Tokamak Project. In addition, the particle acceleration on the Cray C90 was also measured to determine if optimization efforts on the RISC processors degraded performance on the vector processors. (Charge deposit on the C90 was not

measured, since the algorithm being tested here does not vectorize.) The compilers used were if77 (version 5.0.1) on the Paragon, cft77 (version 6.2.0.9) on the T3D, xlf (version 3.01) on the S1'2, and cft77 (version 6.0.4.9) on the C90. The compiler options which consistently gave the best performance was used. These were:

Intel Paragon: if77 -04 -Mr8 -Mr8intrinsics -nx
Cray T3D: cf77 -01
IBM S1'2: xlf -O3 -qarch=pwr2 -qautodbl=dblpad4

Peak single processor speeds for the machines are: 75 Mflops for the Paragon, 150 Mflops for the T3D, 266 Mflops for the S1'2, and 950 Mflops for the C90. For the tests done here, an IBM RS/6000 model 590 workstation was used, which is identical to the so-called "wide node" version of the S1'2 processor.

Case 1: Initial code

Appendix 1 shows the Fortran listing of the 2d charge deposit subroutine which will be used as the starting point for optimization. It minimizes the number of floating point operations and uses a data layout which represents current typical practice for a. scalar processor. Specifically, the components of the particle coordinates are kept in separate arrays, X, Y, VX, and VY. The subroutine assumes periodic boundary conditions and has additional checks to ensure addresses are properly wrapped around. There are 20 multiplies, 19 adds, 2 convert to integer and 2 convert from integer operations, 11 loads and 9 stores in this subroutine. Since conversions to and from integers generally make use of the floating points units, they will be counted as one FLOP (floating point operation) each. Ignoring other integer operations, there is a total of 43 FLOPs per particle for the charge deposit.

Appendix 2 shows a similar listing for the acceleration subroutine. The components of the electric field are kept in separate arrays, FX and FY. An additional kinetic energy calculation is included, which is used to check code integrity. There are 40 multiplies, 34 adds, 2 convert to integer, 2 convert from integer, and 4 floating point comparisons, as well as 22 loads and 4 stores in this subroutine, which gives a total FLOP count of 82 per particle.

The benchmark results for the deposit are summarized in Table I, for the acceleration in Table 11, and the total (acceleration plus deposit) are in Table 111. In this table we will give the speed in MFLOPs/Second. In order to make a meaningful comparison of optimization across processors, we will also give the per cent of peak speed obtained. Note that for the initial case, one obtains about 9% of peak speed for the Paragon and T3D, and 21% for the S1'2. We are also getting about 23% of peak for the acceleration on the C90.

V. General Optimization Strategies

In this section we will test various strategies which improve effective use of cache and arithmetic pipelines. In the following sections, we will introduce one

change at a time. Unless otherwise noted, the changes will be cumulative, that is, when we discuss changes for Case 4, it will also implicitly include the changes already made in Cases 3 and 2.

Case 2: Storing particle components together

The first thing we can try is to reduce the distanced memory references in the particle array. If the particle coordinates are stored in separate arrays, then four cache lines must be brought into memory to fetch them. Storing them together in a two dimensional array PART, as follows:

```
PART(1, j) = X(j)
PART(2, j) = Y(j)
PART(3, j) = VX (j)
PART(4, j) = VY (j)
```

mean.s that al l four coordinates are likely to reside in one cache line (depending on alignment) and thus may be brought into the cache together. This change does not require any additional memory usage, just reorganization.

The results of this change (shown in Tables 1-111) indicate that the acceleration improves on all the RISC processors and degrades slightly on the C90. The deposit degrades slightly on the S1'2 and Paragon, and improves slightly on the T3D. The probable reason for the deposit degrading is that the VX and VY coordinates are brought into cache but are not used in the deposit subroutine. The C90 degrades because the original data layout for the particles had an optimum stride 1 when vectorized, but not in the new layout. Overall, performance improved in the range of 5-20%.

Case 3: Storing particle and field components together

The next thing we can try is to reduce the distance of memory references in the field array, by rearranging the field data as follows:

```
FXY(1, j,k) = FX(j, k)
FXY(2, j,k) = FY(j, k)
```

where (j,k) refers to the grid in two dimensions. This change also does not require any additional memory usage. One finds that the acceleration improves on all the RISC processors and is unchanged in the C90. This change does not effect the deposit. Overall, performance improved from Case 2 by an additional 5-18%.

Case 4: Removing IF statements by index arrays

The pipeline architecture of the RISC processors can make IF statements relatively expensive, since it could potentially inhibit the filling of arithmetic pipelines. So the next thing to try is to remove 1 F statements which occur when

ensuring addresses are properly wrapped around. One way this can be done is by storing a precalculated array of indices , for example:

```
dimension  np(nx)
rip(j) = j+1
if (j. eq. nx) rip(j) = 1
```

and then introduce another layer of indirection, for example in the charge deposit;

```
n = int (x)
q(np (n)) = q(np (n) ) + qm*dx
```

This is often done in vectorizing code on the Cray. However, it introduces another array which must fetched and competes for space in the cache. This change requires small additional memory usage. The results of this change indicate that timings improve from Case 3 by 5-10% on all processors except for the Paragon.

Case 5: Removing IF statements by using guard cells

Alternatively, one could use extra guard cells to allow access to data beyond the boundaries, and then add them up or replicate them outside the particle subroutines. For example, with linear interpolation, the force arrays are replicated before the push by enlarging the field array to include one extra guard cell on the right in each dimension, as follows:

```
dimension fxy(2, nx+1, ny+1)
fxy(1, j,k)      = fx(j, k)
fxy(1, nx+1, k) = fx(1, k)
fxy(1, j,ny+1) = fx(j,1)
```

and similarly for fy(j,k). The charge density is combined after the deposit as follows:

```
dimension q(nx+1,ny+1 )
q(1, k) =  q(1, k) + q(nx+1  k)
q(j, l) =  q(j,1) + q(j, ny+1)
```

Such use of guard cells is also used in the parallel version of this code to avoid off-processor memory reference.

With quadratic interpolation, one needs to enlarge the field arrays to include 3 guard cells, one on the left and two on the right in each dimension. For example one can store the x component of the field array as follows:

```
dimension fxy(2, nx+3, ny+3)
fxy(1, j+1, k+1)   = fx(j, k)
fxy(1, 1,k+1)      = fx(nx, k)
fxy(1, nx+2, k+1) = fx(1, k)
fxy(1, nx+3, k+1) = fx(2, k)
```

```
fxy(1,j+1,1)      = fx(j,ny)
fxy(1,j+1,ny+2) = fx(j,1)
fxy(1,j+1,ny+3) = fx(j,2)
```

and similarly for fy and q.

These changes do require some additional memory usage. Furthermore, it may require modification to the field solver (or the use of temporaries) to handle the modified data structure. The result of adding this change to the previous changes (instead of using index arrays) indicate that this method of removing IF statements is faster on all processors than the use of index arrays. Therefore index arrays will no longer be considered. Overall, timings improve compared to Case 3 by 18-45%. except on the Paragon.

Incorporating all the changes represented by Case 5 (storing particle and field components together, removing IFs with guard cells) now gives speeds that are 10% of peak on the Paragon, 19% of peak on the T3D, and 27% of peak on the S1'2. In addition, we are getting 38% of peak for the acceleration on the C90. Compared to the original version, one finds that the total execution speed on the T3D more than doubles, improves about 30% on the S1'2, and 14% on Paragon. Furthermore, acceleration on the C90 has improved 65%.

VI. Processor Specific Optimization Strategies

The strategies we have used until now have been general ones, which work in almost all cases. However, at this point, it was no longer obvious to us how to obtain even better performance. We decided, therefore, to examine the behavior of individual processors in more detail by studying the assembly output of each compiler. By comparing this with what we knew of each processor's architecture, we could then attempt to determine what bottlenecks existed and what changes could be made to the Fortran to improve performance. The following sections, therefore discuss each processor individually, although we are still interested in discovering general strategies, if possible.

We will start the investigations using Case 5 as a baseline, but with one additional modification. Specifically, we changed the interpolation so that the adjacent field elements were accessed in memory in a monotonically increasing order, and the weights were calculated in the order they were used in the interpolation. By itself, this additional modification to Case 5 made no noticeable difference in performance. However, when used in conjunction with the additional changes discussed below, the modifications were beneficial.

A. Intel Paragon

The Intel Paragon uses the Intel 80860/X1' (i860) processor with a 50 MHz clock [5]. It has one fixed point unit, and two floating point units (one adder and one multiplier). in double precision, it can produce one add every clock and one multiply every two clocks, for a maximum performance of 75 MFlops. integer

arithmetic and floating loads/stores can occur simultaneously with floating point arithmetic. Floating point adds take 3 clocks to complete and multiplies take 6 clocks. There are 32 integer registers and 32 single precision floating point registers which can be configured as 16 double precision registers. There is no hardware divide or square root, although there are instructions for 7-bit accurate divides and reciprocal square roots. The cache is four-way set-associative, 16 KB in size with 32 byte cache lines.

There are both pipelined and non-pipelined versions of the add, multiply and load instructions. The pipelined loads, however, go directly from memory to register and bypass the cache.

We have already noted that the performance of the Paragon improved very little in response to the changes we have made. The reason turned out that the compiler did not use any of the pipelined instructions available to it. Since no overlapping and pipelining was used, the performance is dominated mainly by the number of operations, which has remained the same. The inhibition to the use of pipelining seems to be the indirect addressing used in these subroutines.

Other features were noted. Since multiples take twice as long as adds, peak speed can only be achieved when there are twice as many adds as multiplies. This is not the case here (and probably is rarely true). In double precision, there are only 16 floating point registers available, so that the compiler is forced to keep many temporaries in memory. These are architectural features one cannot do much about, but they degrade performance.

Case **6:** Using one dimensional addressing

It was also noted that in the i860 processor, the integer and floating point registers are connected and conversions between integer and floating point are done quite efficiently. 1 lowever, two dimensional addressing is quite inefficient. Therefore one possible improvement is to do explicit one dimensional addressing. That is, instead of using an array dimensioned q(nxv,nyv) and referring to element q(i,j), use the array dimensioned q(nxv*nyv), and refer to the element q(nxv*(j-1)+i). When this change is made, one finds the code improves compared to Case 5 by 10% on the Paragon, but degrades 3-5% on the other processors.

B. Cray T3D

The Cray T3D uses the EV4DEC Alpha processor with a 150 MHz clock [6-8]. It has one fixed point and one floating point unit, and can produce one integer add and one floating point calculation (add or multiply) per clock for a maximum performance on 150 MFlops. integer arithmetic and floating loads/stores can occur simultaneously with floating point arithmetic. Integer adds and loads from cache take 3 clocks to complete, while floating point add and multiply pipelines have a 6 cycle delay. integer multiplies take 23 clocks to complete. There are 32 integer and 32 floating point registers. There is a hardware divide which is not pipelined and takes 61 clocks to complete. The cache is direct mapped, 8 KB in size, with 32 byte

cache lines. It takes 22. clocks to load a cache line from memory with a page hit (12 clocks if the data already resides in a read ahead buffer).

Case 7: Integer conversion precalculation

Upon examining the assembler output from the T3D compiler, one feature that was evident was that conversions between floating point and integer data types was relatively time consuming, and the calculation was stalled waiting for these to be done. This conversion not only involved use of the floating point units to truncate and normalize, but there was no direct communication between integer and floating point registers, so that additional loads and store were required. Thus we decided to precalculate the conversions, that is, on entry to the loop, the nearest gridpoint and deviation was found for particle j+1, and result saved for the next iteration, while the rest of the calculation worked on particle j. The results showed improvement from Case 5 by 7 and 20% on the T3D and the S1'2, respectively, but had no effect on the Paragon. This change was disastrous the the C90, however, since the vector compiler could no longer vectorize the loop.

Case 8: Unrolling deposit loop

In examining the performance of the T3D, one notes that the acceleration is substantially more efficient than the deposit. Since the deposit has relatively less calculation and the DEC Alpha processor has relatively deep floating point pipelines (6 cycles) compared to the other processors, it was hypothesized that there was not enough calculation exposed to the compiler. Therefore we modified the code to deposit the charge of two particles simultaneously, where we calculate the address and weight of one particle while depositing the other and then reversing the roles. This can be considered as an extension of Case 7. The result showed that compared to Case 7, this improved the performance of the deposit on the Cray T3D by 12%, but made things worse on the other processors.

C. IBM S1'2

The IBM S1'2 uses the POWER2 processor that is found on the RS/6000 model 590 workstation with a 66.5 MHz clock [9-11]. This processor is an "aggressive superscalar" design that has parallel processing elements and can produce multiple results per clock. It has two floating point units, each of which can produce one multiply and add combination per clock, for a maximum performance of 4 Flops per clock, or a total of 266 Mflops. In addition, the processor has two fixed point units which independently perform integer arithmetic and addressing. The processor is capable of performing up to 4 loads and stores in parallel with computation in a clock period. The floating point multiply and add pipelines have a 2-3 cycle delay. Integer multiplies have a 2 cycle delay. There are 32 integer and 32 floating point registers. There is a hardware divide and square root, which take 19 and 27 cycles to complete, respectively. The cache is four-way set-associative, 256 KB in size with 256

byte cache lines. It takes 21-27 clocks to fill a cache line from memory.

Case 9: 1 d addressing and separation of loads and stores

It was previously noted that precalculating the integer conversions (Case 7) also improved performance on the S1'2, so we will start with that. In addition, using one dimensional addressing (as in Case 6) turned out to improve performance when used in conjunction with integer conversion precalculation. (This was somewhat surprising since one dimensional addressing without integer preconversion (Case 5) was worse on the S1'2.)

The charge deposit subroutine (Appendix 1) deposits charge on the nine nearest grid points to the particle's location. In examining the compiler output, it was noted that each one of the nine deposits was completed before the next was done. This was because the compiler could not be certain that some of the nine deposits would go to the same memory location, so it generates safe code, although the algorithm is designed so that this never happens (especially when guard cells are used), '10 help the compiler recognize that the loads and stores can be overlapped, we loaded all nine deposits into temporaries, then deposited them. The result of both of these changes show that compared to Case 7, performance improved by 13% cm the S1'2 and did not change much on the other processors.

In examining all these processor specific cases, we come to the conclusion that Case 9 gives the best overall results so far. To summarize, Case 9 consists of the following additions to the modified baseline Case 5: integer conversion precalculation, 1 dimensional addressing and separation of loads and store in the deposit. This case gives the best result on the S1'2 (37% of peak), and the second best on the T3D (21 % of peak) and Paragon (1 O% of peak). However, this version should be not used on the Cray C90 because it fails to vectorize. The code listing for the deposit for this case is shown in Appendix 3. The acceleration is quite similar.

VII. l'article Sorting

One important feature we have not concentrated on is cache reuse. in a PIC code maximum cache reuse occurs when all the particles in the same cell are processed together. It turns out (the)this was the way the code was initialized. Thus by measuring the degradation of performance as the particle randomize, one can estimate the importance of cache reuse in this case. The results for Case 9, show that the difference in performance between sorted particles and randomized particles is about 1.24 on the Paragon, 1.71 on the T3D and 1.0 on the S1'2. The T3D results as a function of time are shown in Figure 1 for illustration. These results reflect the size of cache: for the test problem wc are considering, one field array is 8 times bigger than the cache size on the T3D, whereas it is one fourth the cache size on the SP2.

Thus if particles were kept sorted, one could expect a further speedup of 24% on the Paragon, 70% on the T3D, and no improvement on the S1'2. Interestingly enough, keeping particles sorted degrades performance about 46% on the C90 (for Case 5). This is because processing particles at the same grid point causes bank

conflicts in memory access on that machine.

One side effect of sorting is that initial particle identities are lost. For example, if one had initialized the first 100 particles in a special way, one no longer knows where they are in the particle array after sorting, unless one keeps track of their original location in a separate array.

Case 10: Simple bin sort with Case 9

Since particles do not have to be exactly sorted to obtain benefits of cache reuse, one approach is to sort them only once in a while, and keep them only approximately sorted. A simple bin sorting routine was written which calculates how many particle there are at each grid point and their locations in the particle array. The particle data is then copied in grid order using the location array and another temporary array. This sorting was used in conjunction with Case 9.

The time to sort was approximately the same as the time to advance the particles one time step, and it was determined empirically that sorting every 25 time steps was optimum, Thus sorting incurred a 4% overhead. It turned out that sorting in both x and y was unnecessary; sorting in y alone gave slightly better performance.

The results in Table 111, Case 10, are time averages over the entire run and include the overhead of sorting. The T3D improved to 32% of peak (48 MFlops), and the Paragon to 12% of peak (9 M Flops). The S1'2 degraded slightly. The T3D results as a function of time are also illustrated in Figure 2. A more efficient sorting routine would give only slight additional improvements.

Versions of the code were also constructed using linked lists to deposit charge from all the particles into a given cell at once while leaving the particles in place in their array. This was not found to be competitive when the cost of constructing and maintaining the linked list was included.

VIII. Conclusions

We have found that optimization strategies for RISC processors fall into two main categories, relating to effective usc of cache and arithmetic pipelines. in Cases 2 and 3, cache use was improved by avoiding large strides in memory and by using data stored in contiguous memory locations as much as possible. In Case 10, cache reuse was improved by occasional sorting of particles. More effective use of arithmetic pipelines was achieved by removing IF statements, reordering the calculation, precalculating integer conversions, making use of one dimensional addressing, and separation of loads and stores in the charge deposit.

There was no magic bullet. Each of these changes improved performance by small amounts, but when aggregated, substantial improvement was achieved. The best version for RISC processors was Case 10. Compared to the initial version, this case improves performance by 343% on the T3D, by 78% on the S1'2, and by 38% on the Paragon.

Acknowledgments

Access to the Intel Paragon at the Jet Propulsion Laboratory, Pasadena, California, was provided by NASA's Office of Aeronautics. We wish to acknowledge assistance from Edith Huang from JPL.

Access to the IBM RS/6000 Model 590 was provided by the Office of Academic Computing, UCLA, and we wish to acknowledge assistance provided by Paul Hoffman.

Access to the Cray C90 and T3D was provided by the National Energy Research Supercomputer Center, in Livermore, California.

The research by V.K.D. was carried out in part at UCLA and was sponsored by USDOE and NSF. It was also carried out in part at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The research of P.C.L. and S.R.K. was supported by AFOSR (Contract #F49620-94-I -0336), NSF (Cooperative Agreement CCR-91 20008), and USDOE.

References:

[1] J. M. Dawson, V. 1 Decyk, R. Sydora, and P. Liewer, "High-Performance Computing and Plasma Physics," Physics Today, vol. 46, no. 3, p. 64 (1993).

[2] C. K. Birdsall and A. B. Langdon, Plasma Physics via Computer Simulation [McGraw-I Iill, New York, 1981].

[3] **1'.** C. Liewer and V. K. Decyk, "A General Concurrent Algorithm for Plasma Particle-in-Cell Codes," J. Computational Phys. 85, 30? (1989).

[4] V. K. Decyk, "Skeleton PIC Codes for Parallel Computers," Computer Physics Communications, 87, 87 (1995).

[5] i860 64-Bit Microprocessor Programmer's Reference Manual, Intel Corporation, Santa Clara, California, 1990.

[6] Alpha Architecture Handbook, Digital Equipment Corporation, 1992.

[7] Jeff Brooks, "Single PE Optimization Techniques for the CRAY T3D System", Cray Research Corporation, 1995.

(8] Alice E. Koniges and Kevin R. Lind, "Optimization Techniques and Early Results for T3D Performance on industrial and Academic Applications," Computers in Physics 9, 399 (1995).

[9] S. W. White and S. Dhawan, "POWER2: Next generation of the RISC System/6000 family," IBM J. Res. Develop. 38, 493 (1994).

[10] R. C. Agarwal, F. G. Gustafson, and M. Zubair, "Exploiting functional parallelism of POWER2 to design high-performance numerical algorithm s," 1 BM J. Res. Develop. 38, 563(1 994).

[11] Optimization and Tuning Guide for Fortran, C and C++, IBM Publication SC09-1705-00, 1993.

Table 1 : Deposit Benchmark in Mf 1 ops (% of peak)

Case	IBM SP2	Cray T3D	lnt c1 Paragon
1 initial	64 (24%)	14 (9%)	6.6 (9%)
2 particle array	61 (23%)	15 (10%)	6.4 (9%)
3 field array	61 (23%)	16 (11%)	6.4 (9%)
4 index array	74 (28%)	16 (16%)	6.0 (8'%)
5 guard cells	83 (31%)	24 (16%)	6.4 (9%)
61d addressing	81 (30%)	21 (14%)	7.3 (10%)
7 precalculat ion	88 (33%)	24 (16%)	6.6 (9%)
8 loop unrol 1 ing	61 (23%)	27 (18%)	6,3 (8%)
9 best. case	101 (38%)	25 (1-7%)	6.3 (8%)

Table II : Acceleration Benchmark in Mflops (% of peak)

Case	IBM SP2	Cray T3D	Intel Paragon	Cray C90
1	51 (19%)	14 (9%)	6.4 (9%)	2,15 (23%)
2	57 (21%)	18 (12%)	7.0 (9%)	210 ('22%)
3	63 (24%)	24 (16%)	8.0 (11%)	210 (22%)
4	66 (25%)	25 (17%)	7.3 (10%)	29? (31%)
5	69 (?6%)	33 (22%)	7.9 (11%)	363 (38%)
6	66 (25%)	31 (21%)	8.6 (11%)	358 (38%)
7	86 (32%)	36 (24%)	7.8 (10%)	46 (5%)
8	84 (32%)	36 (24%)	7.8 (10%)	47 (5%)
9	96 (36%)	36 (24%)	8.4 (11%)	42 (5%)

Table III: Total Benchmark in Mflops (% of peak)

Case	IBM SP2	Cray T3D	Intel Paragon
1 initial	55 (21%)	14 (9%)	6.5 (9%)
2 particle array	59 (22%)	17 (11%)	6.8 (9%)
3 field array	62 (23%)	20 (13%)	7.4 (10%)
4 index array	68 (26%)	21 (14%)	6.8 (9%)
5 guard cells	73 (27%)	29 (19%)	7.4 (10%)
6 1d addressing	71 (27%)	27 (18%)	8.1 (11%)
7 precalculat ion	87 (33%)	31 (21%)	7.4 (10%)
8 loop unrolling	74 (28%)	32 (21%)	7.2 (lo%)
9 best case	98 (37%)	31 (21%)	-/.5 (10%)
10 sort ing	9-/ (36%)	48 (32%)	9.0 (12%)

Appendix 1 : Initial deposit subroutine

```
      subroutine dpost ? (x, y,q, qm, nop, nx, ny)
      dimension x(nop) , y(nop) , q(nx ,ny)
      qmh = . 5*qm
      do 10 j = 1, nop
c find interpolation weights
      nn =  x(j) + .5
      dxl = x(j) - float (nn)
      nn = nn + 1
      if (nn. gt. nx) nn = nn - nx
      amx = qm* (.75 - dxl*dxl)
      np = nn + 1
      if (rip. gt .nx) np = np - nx
      dxp = qmh* (.5 + dxl) **2
      nl = nn - 1
      if (nl. lt .1) nl = nl + nx
      dxl =- qmh* (.5 - dxl )"?.
      mm== y(j) + .5
      dyl = y(j) - float (mm)
      mm = mm + 1
      if (mm. gt. ny) mm = mm - ny
      amy = .75 - dyl*dyl
      rnp = mm + 1
      if (rep. gt .ny) mp = rnp - ny
      dyp = . 5* (.5 + dyl) **2
      ml "mrn- 1
      if (nil.lt_.1) ml = ml + ny
      dyl '- .5* (.5 - dyl)**2
cdeposit charge
      q(nn, mm) = q (nn, mm) + amx*amy
      q (rip, mm) = q (rip, mm) + dxp*amy
      q(nl, mm) = q(nl, mm) + dxl*amy
      q(nn, mp) = q(nn, mp) + amx*dyp
      q(np, mp) = q (rip, mp) + dxp*dyp
      q(nl, mp) = q(nl, mp) + dxl*dyp
      q(nn, ml) = q(nn, ml) + amx*dyl
      q(np, ml) = q(np, ml) + dxp*dyl
      q(nl, ml) = q(nl, ml) + dxl*dyl
   10 continue
      return
      end
```

Appendix 2: Initial push subroutine

```
      subroutine push2 (x, y,vx, vy, fx, fy, qtm, dt ,ek, nop, nx, ny
     1)
      dimension x(nop) , y(nop) , vx(nop) , vy(nop)
      dimension fx(nx, ny) , fy(nx, ny)
      zero = 0.
      anx = float, (nx)
      any = float (ny)
      sum] ' 0.
      do 10 j = 1, nop
c find interpolation weights
      nn = x(j) + .5
      dx = x(j) - float (nn)
      nn = nn + 1
      if (nn. gt. nx) nn =- nn    nx
      amx = . 75 - dx*dx
      np = nn + 1
      if (np. gt. nx) np = np -- nx
      dxp = .5* (.5 + dx) **2
      nl = nn - 1
      if (nl. lt .1) nl = n] + nx
      dxl = .5* (.5 - dx) **2
      mm = y(j) + .5
      dy = y(j) - float (mm)
      mm = mm + 1
      if (mm. gt. ny) mm = mm - ny
      amy = . 75 - dy*dy
      mp = mm + 1
      if (mp. gt. ny) mp = mp - ny
      dyp = . 5* (.5 + dy) **2
      ml = mm - 1
      if (ml. lt .1) ml = ml + ny
      dyl =  .5* (.5 - dy) **2
c find acceleration
      dx = amy* (amx*fx (nn, mm) + dxp*fx(np, mm) +dxl *fx(nl ,mm) )
     1 4    dyp* (amx*fx (nn, mp) +dxp* fx(np, mp) +dxl *fx(nl, mp) )
     2 +    dyl* (amx*fx (nn, ml) +dxp*fx (np, ml) +dxl *fx(nl, ml))
      dy = amy* (amx*fy (nn ,mm) +dxp*fy( np,mm) +dxl *f y(nl ,mm) )
     1 +    dyp* (amx*fy (nn, mp) +dxp*fy (rip, mp) +dxl *fy(nl, mp) )
     2 +    dyl* (amx* fy(nn, ml) +dxp*f y(np, ml) +dxl *fy(nl, ml))
c new velocity
      dx = vx (j) + qt m*dx
      dy = vy (j) + qtm*dy
c average kinetic energy
      suml =  suml + (dx + vx(j)) **2 + (dy + vy(j)) **2
      vx (j) = dx
      vy (]) = dy
c new posi ti on
      dx = x(j) -t dx*dt
      dy = y(j) + dy *dt
c per i odi c boundary conditions
```

```
      if (dx. lt.zero) dx = dx + anx
      if (dx. ge. anx) dx = dx - anx
      x(j) = dx
      if (dy. lt. zero) dy = dy + any
      if (dy. ge. any) dy = dy - any
      y(j) = dy
   10 continue
c normalize kinetic energy
      ek = ek + .125*sum1
      return
      end
```

Appendix 3: Final deposit subroutine

```
      subroutine dpost2 (part. ,q, qm, nop, idimp, nxv, nxyv)
      dimension part (idimp, nop) , q(nxyv)
c begin first particle
      nnn = part (1,1) + .5
      mmn = part (2,1) + .5
      dxn = part (1 , 1 ) - f loat (nnn)
      dyn = part (2, 1) - float (mmn)
      qmh = .5 * qm
c find interpolation weights
      do 10 j =2, nop
      nn = nnn + 1
      mm "' nxv*mmn
      nnn = part(1 , j) + .5
      mmn = par t(2, j) + .5
      dxp = dxn
      dyp = dyn
      dxn = part (1, j) -- float (nnn)
      dyn = part (2, j) - float_ (mmn)
      amx = qm* ( .75 - dxp*dxp)
      amy = . 75 -- dyp*dyp
      ml = mm + nn
      dxl = qmh* ( (.5 - dxp) **2
      dxp = qmh* (.5 + dxp) **2
      mn = ml + nxv
      dyl = .5* (.5 - dyp)**2
      dyp = . 5' (.5 + dyp)**2
      rnp = mn + nxv
c deposit charge
      dx = q(mn) + dxl*amy
      dy = q(mn+1) + amx*amy
      amy = q(mn+2) + dxp * a my
      dxl = q(m)) + dxl*dyl
      dyl = q(ml +1) + amx*dyl
      dyl = q(ml +2) + dxp*dyl
      dxl = q(mp) + dxl *dyp
      amx = q (mp+ 1 ) + amx*dyp
      dyp = q (mp+ 2) + dxp*dyp
      q(mn) = dx
      q(mn+1) = dy
      q(mn+ ?.) = amy
      q (ml ) = dx1
      q(ml+1) = dyl
      q(ml +2) = dyl
      q (rep) = dxl
      q (mp+ 1 ) = amx
      q(mp+2) = dyp
   10 continue
c deposit charge for last particle
      nn = nnn + 1
      mm = nxv*mmn
```

```
      amx =  qm* ( .-/5 - dxn*dxn)
      amy =  .75 - dyn*dyn
      ml = mm + nn
      dxl = qmh*(.5 - dxn)**2
      dxp = qmh*(.5 + dxn)**2
      mn = ml + nxv
      dyl =  .5* (.5 - dyn)**2
      dyp =  .5* (.5 + dyn)**2
      mp = mn + nxv
c deposit charge
      q(mn) = q(mn) + dxl*amy
      q(mn+1) =  q(mn+1 ) + amx*amy
      q(mn+2) = q(mn+2) + dxp*amy
      q(ml ) = q(ml) + dxl*dyl
      q(ml+1) =  q(ml+1) + amx*dyl
      q(ml+2) = q(ml+2) + dxp*dyl
      q(mp) =  q(mp) + dxl*dyp
      q(mp+1) = q (mp+1)+ amx*dyp
      q(mp+2) = q (mp+2) + dxp*dyp
      return
      end
```

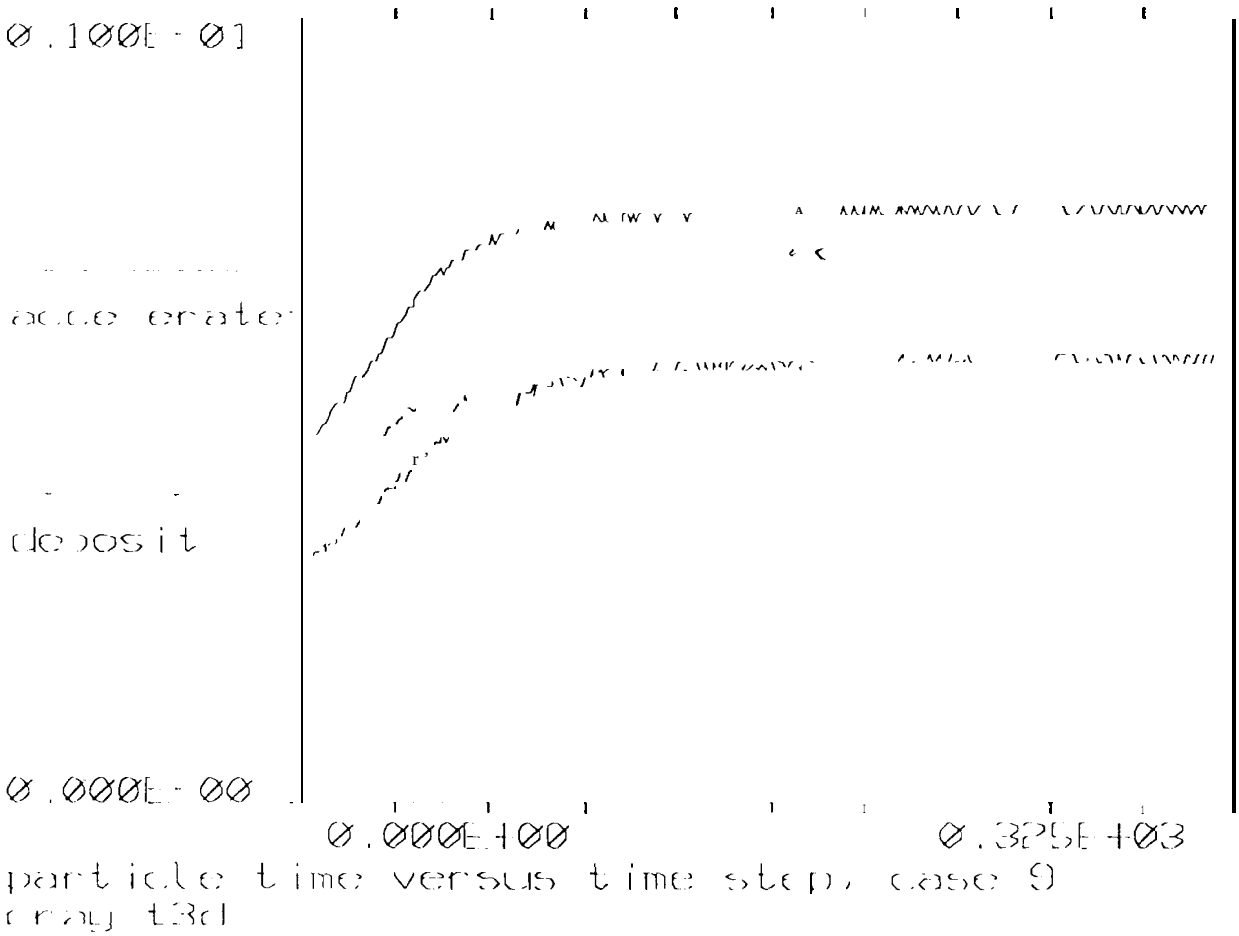

Figure 1

CPU time in seconds for 327, 680 particles as a function of time step for acceleration and deposit steps. Curve shows slowdown in CPU time as particles become randomized.

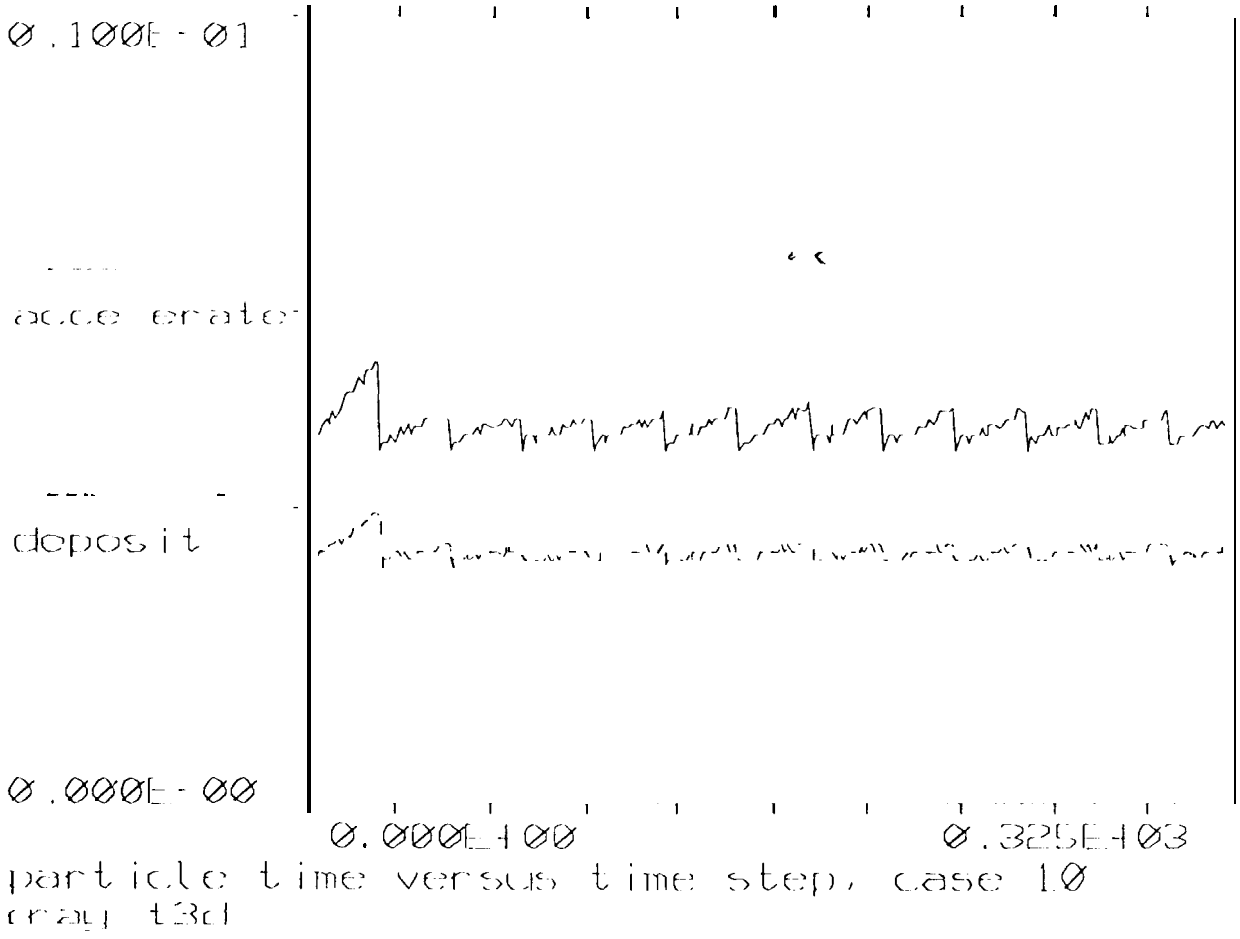

Figure 2

CPU time in seconds for 327, 680 particles as a function of time step for acceleration and deposit. steps. Curve shows slowdown in CPU time, is prevented by resorting particles every 25 time steps.